Towards Imperative Type Systems

AUTHOR(S):

Otake, Kazuo

CITATION:

Otake, Kazuo. Towards Imperative Type Systems. 数理解析研究所講究録 1993, 851: 127-138

ISSUE DATE:

1993-10

URL:

http://hdl.handle.net/2433/83700

数理解析研究所講究録
第 851 巻 1993 年 127-138

Towards Imperative Type Systems

Kazuo Otake

(otake@swl.cl.nec.co.jp)
C&C Research Laboratories
NEC corporation.

Abstract

We developed an imperative reduction system called i-calculus which has imperative type system. i-calculus has a representation of variable and assignment. The type system has a representation of effects on variable that is allocation and value reading and assignment. For an example of merit of imperative calculation model, We show an implementation of imperative record type It is also shown some relation between imperative type and program static analysis.

1 Introduction

Type theory is successful in functional programming that is essentially based on λ calculus. Consequently, many good results of type theory in functional programming are not applicable to imperative programming. There are many reason that imperative programming is important. First of all, the great portion of real world software is written in imperative programming languages. Another reason is that there are lot of works on the area of object oriented programming and type theory. However, almost such works are done on functional object oriented programming. Since object is an entity that can change its internal state, the notion of object oriented computing can be well captured by a imperative computing.

One easy way of amendment is to introduce variable type (for example, *ref* type in ml) and ignore effects on the variable. This is a good solution for programs almost functional, but for programs almost imperative. Our goal is a type system that can represent effects on variables such as assignment. We also need a calculation system with reduction. Because, a lot of impressive results of type theory is related to reduction (such as normalization theorem or formula as a type and normalization of proof as a reduction principle).

We achieved such goal by introducing a term that represents imperative effect. This term works as a bridge between imperative calculation and imperative type.

2 Some works on Imperative calculus and imperative type systems

There are many works on calculus with assignment and imperative type systems. Some of them are:

FX by Gifford et al (1987)
Call by Value, Effect Type system, No Reduction Semantics

Forthys by Reynolds (1987)
Call by Name, Two-phase(Reduction + Execution) Semantics, Intersection Type

λ-v-CS by Felleisen (1988)
Call by Value, Reduction Semantics, Untyped

Monad comprehension by Wadler (1990)

Single-threaded polymorphic lambda by Guzman et al (1990)

ILC(Imperative Lambda Calculus) by Swarup et al (1991)
Call by Name, Reduction Semantics

λ_{var} by Odersky et al (1992)
Call by Name, Reduction Semantics, Untyped

Only FX system has types of effect on variable, however FX doesn't have reduction. In the following section we see FX system and C-rewriting system briefly.

2.1 FX system

Kernel of FX system[1] is shown below. FX kernel is essentially a second order polymorphic lambda calculus with re-writable memory region.

Kind ::= Region | Effect | Type

$\rho \in$ Region $\quad \tau \in$ Type $\quad \epsilon \in$ Effect

Effect ::= Dvar | (ALLOC ρ) | (READ ρ) | (WRITE ρ) | (MAXEFF ϵ^*) | PURE

Region ::= RCONST | Dvar | (UNION $Region^+$)

Type ::= Dvar | (SUBR $(\tau)\epsilon\tau$) | (POLY (Dvar: Kind)$\epsilon\,\tau$) | (REF $\rho\tau$)

FX term has a piar of separete types.

e : Type ! Effect

One is a term type and the other is a effect type. FX inference rule is shown below.

(abstraction)

$$\frac{A[x \leftarrow \tau], B \vdash e: \tau' ! \epsilon}{A, B \vdash (\lambda(x:\tau).e): (\mathrm{SUBR}(\tau)\epsilon\tau') ! \mathrm{PURE}}$$

(application)

$$\frac{A, B \vdash e_1: (\mathrm{SUBR}(\tau_1)\epsilon\tau_2) ! \epsilon_1 \quad A, B \vdash e_2: \tau_1 ! \epsilon_2}{A, B \vdash (e_1 e_2): \tau_2 ! (\mathrm{MXF}\,\epsilon_1\epsilon_2\epsilon)}$$

(polymorphic abstraction)

$$\frac{A, B[d \leftarrow \kappa] \vdash e: \tau ! \epsilon \quad \forall x \in FV(e).d \notin FV(A(x))}{A, B \vdash (\Lambda(d:\kappa).e): (\mathrm{POLY}(d:\kappa)\epsilon\tau) ! \mathrm{PURE}}$$

(polimorphic application)

$$\frac{\mathrm{A,B} \vdash e\colon (\mathrm{POLY}(d\colon \kappa)\epsilon'\tau')!\,\epsilon \quad \mathrm{B} \vdash \delta\colon \kappa}{\mathrm{A,B} \vdash (\mathrm{PROJ}\, e\delta)\colon \tau'[\delta/d]!\,(\mathrm{MXF}\,\epsilon\epsilon'[\delta/d])}$$

(allocation)

$$\frac{\mathrm{B} \vdash \rho\colon \mathrm{REGION} \quad \mathrm{B} \vdash \tau\colon \mathrm{TYPE} \quad \mathrm{A,B} \vdash e\colon \tau!\,\epsilon}{\mathrm{A,B} \vdash (\mathrm{NEW}\,\rho\tau e)\colon (\mathrm{REF}\,\rho\tau)!\,(\mathrm{MXF}\,\epsilon(\mathrm{ALLOC}\rho))}$$

(reading)

$$\frac{\mathrm{A,B} \vdash e\colon (\mathrm{REF}\rho\tau)!\,\epsilon}{\mathrm{A,B} \vdash (\mathrm{GET}\, e)\colon \tau!\,(\mathrm{MXF}\,\epsilon(\mathrm{READ}\rho))}$$

(writing)

$$\frac{\mathrm{A,B} \vdash e_1\colon (\mathrm{REF}\,\rho\tau)!\,\epsilon_1 \quad \mathrm{A,B} \vdash e_2\colon \tau!\,\epsilon_2}{\mathrm{A,B} \vdash (\mathrm{SET}\, e_1e_2)\colon \mathrm{UNIT}!\,(\mathrm{MXF}\,\epsilon_1\epsilon_2(\mathrm{WRITE}\,\rho))}$$

(effect masking)

$$\frac{\mathrm{A,B} \vdash e\colon \tau!\,\epsilon \quad x \in FV(e) \Rightarrow d\colon \mathrm{REGION} \notin FV(A(x)) \quad d\colon \mathrm{REGION} \notin FV(\tau)}{\mathrm{A,B} \vdash e\colon \tau!\,\epsilon[\psi/d]}$$

2.2 C-rewriting system

C-rewriting system[2] is a previous version of successful λ-v-CS[3] by Felleisen. C-rewriting system is a kind of lambda calculus introducing labeled value V^n and assignment to the labeled value.

$$\begin{array}{rcl} x & : & \text{variable} \\ n & : & \text{label} \\ M & ::= & x \mid \lambda x.M \mid \sigma X.M \mid MM \mid V^n \\ V & ::= & \langle\text{closure}\rangle \\ X & ::= & x \mid V^n \end{array}$$

rewriting rule

The notion of context $C[\,]$ is introduced in order to controle the order of calculation.

$$C[\,] ::= [\,] \mid V\,C[\,] \mid C[\,]\,M$$

$$\begin{array}{l} C[(\lambda x.M)V] \longrightarrow_c C[M[V^n/x]] \quad \text{(where } n \text{ is new label)} \\ C[(\sigma U^n.M)V] \longrightarrow_c C[M][n := V] \\ C[V^n] \longrightarrow_c C[V[n := V]] \end{array}$$

$$\begin{array}{rcll} x[n := V] & = & x & \\ U^n[n := V] & = & V^n & \\ U^m[n := V] & = & U[n := V]^m & \text{where } n \neq m \end{array}$$

$$
\begin{aligned}
(\lambda x.M)[n := V] &= \lambda x.M[n := V] \\
(MN)[n := V] &= M[n := V]N[n := V] \\
(\sigma X.M)[n := V] &= \sigma X[n := V].M[n := V]
\end{aligned}
$$

As shown above, assignment is accomplished by rewriting whole context.

$$\cdots 0^n \cdots (\sigma 0^n.M)1 \cdots 0^n \cdots \longrightarrow \cdots 1^n \cdots M' \cdots 1^n \cdots$$

New labeled value is allocated by application.

$$(\lambda x.x)0 \longrightarrow 0^n \longrightarrow 0$$

i-rewriting system

Syntax

x : name

n : label

$e ::= x \mid k \mid V^n \mid \lambda x.e \mid ?e \mid !e \mid \#e \mid ee$

$k ::= ()$

$V ::= k \mid \langle\text{closure}\rangle \mid V^n$

$\langle\text{closure}\rangle \stackrel{\text{def}}{=}$ abstractions(that is $\lambda x.e$) with no free names.

Context

$C ::= [] \mid VC \mid Ce \mid ?C \mid !C \mid \#C$

$C[e] \stackrel{\text{def}}{=}$ a term that is prelaced the hole of C[ ] by e

Rewriting rules

$C[(\lambda x.e)V] \longrightarrow C[e[V/x]]$

$C[?V^n] \longrightarrow C[V[n := V]]$

$C[!U^n V] \longrightarrow C[()][n := V]$

$C[\#V] \longrightarrow C[V^n]$ (n is fresh)

$e[V/x]$: replacing all free x by V in e.
$e[n := V]$: replacing all $\bullet^n$ by V^n in e .

Macro

$\lambda.e \stackrel{\text{macro}}{=} \lambda d.e$ where d is a new name.

Figure 1: i-rewriting system

3 i-rewriting system

In this section, we introduce a rewriting system with assignment and introduce an imperative type system for the system later.

The rewriting system is essectially equivalent to the FX kernel except for cell. Cell is a representation of variables for use of reduction.

Figure 1 shows the systax and rewriting rules of i-rewriting system. Cell V^n represent variables which is a pair of value V and label n. Assignment to a cell $!V^nU$ is acomplished by replacing V^n to U^n in whole of the program executing. Thus, the effect of assignment rewriting rule change the context.

Swap(1)

Let us show an example of rewriting. swap is a routine that exchange the contents of two cells.

$$\text{swap} = \lambda xy.(\lambda.\lambda.\lambda.())(!0^w?x)(!x?y)(!y?0^w)$$

$$\begin{aligned}
&C[\text{swap}\,7^m2^n] \\
&\longrightarrow C[(\lambda.\lambda.\lambda.())(!0^w?7^m)(!7^m?2^n)(!2^n?0^w)] \\
&\longrightarrow C[(\lambda.\lambda.\lambda.())(!0^w7)(!7^m?2^n)(!2^n?0^w)] \\
&\longrightarrow C'[(\lambda.\lambda.\lambda.())()(!7^m?2^n)(!2^n?7^w)] \qquad (\text{where } C' \equiv C[w := 7]) \\
&\longrightarrow C'[(\lambda.\lambda.())(!7^m?2^n)(!2^n?7^w)] \\
&\longrightarrow C'[(\lambda.\lambda.())(!7^m2)(!2^n?7^w)] \\
&\longrightarrow C''[(\lambda.\lambda.())()(!2^n?7^w)] \qquad (\text{where } C'' \equiv C'[m := 2]) \\
&\longrightarrow C''[(\lambda.())(!2^n?7^w)] \\
&\longrightarrow C''[(\lambda.())(!2^n7)] \\
&\longrightarrow C'''[(\lambda.())()] \qquad (\text{where } C''' \equiv C''[n := 7]) \\
&\longrightarrow C'''[()]
\end{aligned}$$

Essential effect of swap is that context C was rewritten to C'''. This effect really depends on the form of C.

4 Effects as terms — Typed i-calculus

i-rewriting system is very close to FX kernel, so FX like typing can be easily introduce to i-rewriting system. For example,

$$?V^n \;:\tau\; !\,(\text{READ}\,\rho)$$

where V^n is a cell in a region ρ . However, there is a new problem in typing i-rewriting system. Since, type is a static property of a program, type should not change by reduction. More precisely, if term e has a type τ and e is rewrited to e' , then e' should has the type τ. However, let us consider the rewriting example below.

$$\begin{aligned}
&?V^n \quad : \tau \;!\; (\text{READ}\,\rho) \\
\longrightarrow\; &V \quad : \tau \;!\; (\text{READ}\,\rho)
\end{aligned}$$

How can we type V as : τ ! (READ ρ) ?

The solution we adopt here is introducing effect term that represent effect on cell. The term $?V^n$ is reduced to something like below and the term has a type below.

$V|\text{EffectTerm} \quad : \quad \tau \ddagger E$

One remark here is the typing introduced here is essentially different from FX typing. In FX typing

$A, B \vdash e : \tau ! \epsilon$

e has a type τ and also has a type ϵ. However, in our typing, term $V|$Effect has a type $\tau \ddagger E$.

We introduce three effect terms related to three effects on cell, that is allocation, reading and writing. The syntax of these effect terms and related rewriting rules are shown below.

[Allocation]

$$\begin{array}{lll} & \#[\tau][\rho]V & : \ \tau^\rho \ddagger (A\rho) \\ \longrightarrow & V^n|\odot^n & : \ \tau^\rho \ddagger (A\rho) \end{array}$$

[Reading]

$$\begin{array}{lll} & ?V^n & : \ \tau \ddagger (R\rho) \\ \longrightarrow & V|\Uparrow^n & : \ \tau \ddagger (R\rho) \end{array}$$

[Writing]

$$\begin{array}{lll} & !\bullet^n V & : \ \mathbf{1} \ddagger (W\rho) \\ \longrightarrow & ()|V\Downarrow^n & : \ \mathbf{1} \ddagger (W\rho) \end{array}$$

In the next section, we give a whole syntax of typed i-calculus (i-rewriting system with typing) and typing rules.

5 syntax of i-calculus

Kind(K) ::= region | effect | type

effect(ϵ) ::= dvar | $(A\rho)$ | $(R\rho)$ | $(W\rho)$ | $(\epsilon \wedge \epsilon)$ | $PURE$

region(ρ) ::= RCONST | dvar | (UNION ρ^+)

type(o) ::= τ | $\tau \ddagger \epsilon$

$\tau ::= \alpha \mid \tau^\rho \mid \Delta$Dvar:Kind.$\tau \mid \tau \rightarrow \tau \ddagger \epsilon$

Type α includes type variable and primitive types such as int. In other words, primitive types are free variables which is globally defined.

n : label

x : name

$e ::= x \mid k \mid V^n \mid \lambda x{:}\tau.e \mid ee \mid \Lambda\alpha{:}K.e \mid e[\tau]$

$k ::= () \mid ? \mid ! \mid \#$

typing rules

[Variable]
$$\Pi, \pi \vdash x : \pi(x)$$

[Cell]
$$\frac{\Pi, \pi \vdash n : \rho \quad \Pi, \pi \vdash V : \tau}{\Pi, \pi \vdash V^n : \tau^\rho}$$

[Abstraction]
$$\frac{\Pi, \pi[x \leftarrow \tau] \vdash e : \tau' \ddagger \epsilon}{\Pi, \pi \vdash (\lambda x{:}\tau.e) : \tau \rightarrow \tau' \ddagger \epsilon}$$

[Application]
$$\frac{\Pi, \pi \vdash e_1 : (\tau \rightarrow \tau' \ddagger \epsilon) \ddagger \epsilon_1 \quad \Pi, \pi \vdash e_2 : \tau \ddagger \epsilon_2}{\Pi, \pi \vdash e_1 e_2 : \tau \ddagger \epsilon \wedge \epsilon_1 \wedge \epsilon_2}$$

[Polymorphic Abstraction]
$$\frac{\Pi[d \leftarrow \kappa], \pi \vdash e : \tau \quad \forall x \in FV(e).d \notin FV(\pi(x))}{\Pi, \pi \vdash (\Lambda d{:}\kappa.e) : (\Delta d{:}\kappa.\tau)}$$

[Polymorphic Application]
$$\frac{\Pi, \pi \vdash e : (\Delta d{:}\kappa.\tau) \quad \Pi \vdash \delta : \kappa}{\Pi, \pi \vdash e[\delta] : \tau[\delta/d]}$$

[Allocation]
$$\frac{\Pi \vdash \rho : \text{region} \quad \Pi \vdash \tau : \text{type} \quad \Pi, \pi \vdash e : \tau \ddagger \epsilon}{\Pi, \pi \vdash \#[\tau][\rho]e : \tau^\rho \ddagger \epsilon \wedge (A\rho)}$$

[Reading]
$$\frac{\Pi, \pi \vdash e : \rho^\tau \ddagger \epsilon}{\Pi, \pi \vdash ?e : \tau \ddagger \epsilon \wedge (R\rho)}$$

[Writing]
$$\frac{\Pi, \pi \vdash e_1 : \rho^\tau \ddagger \epsilon_1 \quad \Pi, \pi \vdash e_2 : \tau \ddagger \epsilon_2}{\Pi, \pi \vdash !e_1 e_2 : \mathbf{1} \ddagger \epsilon_1 \wedge \epsilon_2 \wedge (W\rho)}$$

[Alloc Effect]
$$\frac{\Pi, \pi \vdash n : \rho}{\Pi, \pi \vdash \odot^n : (A\rho)}$$

[Write Effect]
$$\frac{\Pi, \pi \vdash n : \rho \quad \Pi, \pi \vdash V : \tau}{\Pi, \pi \vdash V \Downarrow^n : (W\rho)}$$

[Read Effect]
$$\frac{\Pi, \pi \vdash n : \rho}{\Pi, \pi \vdash \Uparrow^n : (R\rho)}$$

[Type and Effect]
$$\frac{\Pi, \pi \vdash e : \tau \ddagger \epsilon \quad \Pi, \pi \vdash r : \epsilon'}{\Pi, \pi \vdash e|r : \tau \ddagger \epsilon \wedge \epsilon'}$$

[Effect Masking]
$$\frac{\Pi, \pi \vdash e : \tau \ddagger \epsilon \quad \Pi \vdash d : \text{region} \quad x \in FV(e) \Rightarrow d \notin FV(\pi(x)) \quad d \notin FV(\tau)}{\Pi, \pi \vdash e : \tau \ddagger \epsilon[\psi/d]}$$

Definition of FV and fv are shown below.

$$\text{fv}(x) \quad = \quad \{x\}$$

$$
\begin{aligned}
\mathrm{fv}(k) &= \{\} \\
\mathrm{fv}(V^n) &= \{n\} \\
\mathrm{fv}(\lambda x{:}\tau.e) &= \mathrm{fv}(e) - \{x\} \\
\mathrm{fv}(e_1 e_2) &= \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \\
\mathrm{fv}(\Lambda\alpha{:}K.e) &= \mathrm{fv}(e) \\
\mathrm{fv}(e[\tau]) &= \mathrm{fv}(e)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{FV}(\alpha) &= \{\alpha\} \\
\mathrm{FV}(\tau^\rho) &= \{\rho\} \cup \mathrm{FV}(\tau) \\
\mathrm{FV}(\Delta\alpha{:}K.\tau) &= \mathrm{FV}(\tau) - \{\alpha\} \\
\mathrm{FV}(\tau_1 \to \tau_2 \ddagger \epsilon) &= \mathrm{FV}(\tau_1) \cup \mathrm{FV}(\tau_2)
\end{aligned}
$$

6 Reduction of typed i-calculus

Typing rules of i-calculus is designed to move automatically inner effect to outside. For effect terms, we need new set of rewriting rules to move inner effect terms outside. These rules would look like

$$(e|r)d \longrightarrow (ed')|r$$

or

$$d(e|r) \longrightarrow (d'e)|r$$

As indicated above, term d is changed to d' during rewriting. This means that we have a free hand to do something useful during effect moving. One condition is that d and d' should be the same type.

We chose the set of effect term moving rewriting rules as below.

- Allocation

  $$(t|\odot^n)e \longrightarrow (te)|\odot^n$$

  $$U(t|\odot^n) \longrightarrow (Ut)|\odot^n$$

- Writing

  $$(t|V\Downarrow^n)e \longrightarrow (te[n := V])|V\Downarrow^n$$

  $$U(t|V\Downarrow^n) \longrightarrow (U[n := V]t)|V\Downarrow^n$$

  where e[n:=V] is a term replaced all cells $\circ^n$ (that is cell with label n) by V^n.

- $$(t|\Uparrow^n)e \longrightarrow (te)|\Uparrow^n$$

  $$U(t|\Uparrow^n) \longrightarrow (Ut)|\Uparrow^n$$

Using write effect term and above write effect moving rule, it is possible to develop reduction system for i-calculus that no reduction rules change context. (So we call it *i-calculus.*)

The definition of i-calclus is shown below.

(New) context

$$\mathcal{E} ::= [] \mid V\mathcal{E} \mid \mathcal{E}e \mid ?\mathcal{E} \mid !\mathcal{E} \mid \#\mathcal{E} \mid \mathcal{E}|V\Downarrow^n$$

(New) term

$$t ::= \mathcal{E}[e]$$

Where e is the same as in i-rewriting system.

The meanings of the context $\mathcal{E}$ is that there is no effect terms that can affect term in a hole of $\mathcal{E}$.

Reduction rule

$$\mathcal{E}[(\lambda x.e)V] \longrightarrow \mathcal{E}[e[V/x]]$$
$$\mathcal{E}[?V^n] \longrightarrow \mathcal{E}[V[n := V]]$$
$$\mathcal{E}[!U^nV] \longrightarrow \mathcal{E}[()|V\Downarrow^n]$$
$$\mathcal{E}[\#V] \longrightarrow \mathcal{E}[V^n] \quad (n \text{ is fresh})$$

$$(t|V\Downarrow^n)e \longrightarrow te[n := V]|V\Downarrow^n$$

$$U(t|V\Downarrow^n) \longrightarrow U[n := V]t|V\Downarrow^n$$

$$(t|\odot^n)e \longrightarrow (te)|\odot^n$$

$$U(t|\odot^n) \longrightarrow (Ut)|\odot^n$$

$$(t|\Uparrow^n)e \longrightarrow (te)|\Uparrow^n$$

$$U(t|\Uparrow^n) \longrightarrow (Ut)|\Uparrow^n$$

$$?(t|V\Downarrow^n) \longrightarrow ?t|V\Downarrow^n$$

$$!(t|V\Downarrow^n) \longrightarrow !t|V\Downarrow^n$$

$$\#(t|V\Downarrow^n) \longrightarrow \#t|V\Downarrow^n$$

As shown above, no reduction rules rewrite context $\mathcal{E}$. The objective of context $\mathcal{E}$ is to avoid effect conflict. For example, consider the term below.

$$(e|\epsilon)(e'|\epsilon')$$

The result will rewriting will be different between moving the write effect term ϵ first, or moving the write effect term ϵ' first. The definition of i-calculus term

$$t ::= \mathcal{E}[e]$$

exclude such a term with effect conflict and the reduction rules will not generate such term under the control by $\mathcal{E}$.

6.1 Reduction and typing of Swap

Here we show a i-calculus version of reduction of swap. We ommit read effect term in the example for clality.

$$
\begin{aligned}
&\mathcal{E}[\mathrm{swap}\, 7^m 2^n] \\
&\longrightarrow \mathcal{E}[(\lambda.\lambda.\lambda.())(!0^l?7^m)(!7^m?2^n)(!2^n?0^l)] \\
&\longrightarrow \mathcal{E}[(\lambda.\lambda.\lambda.())(!0^l 7)(!7^m?2^n)(!2^n?0^l)] \\
&\longrightarrow \mathcal{E}[(\lambda.\lambda.\lambda.())(()|7\Downarrow^w)(!7^m?2^n)(!2^n?0^l)] \\
&\longrightarrow \mathcal{E}[((\lambda.\lambda.\lambda.())())|7\Downarrow^w\ (!7^m?2^n)(!2^n?0^l)] \\
&\longrightarrow \mathcal{E}[(\lambda.\lambda.())|7\Downarrow^w\ (!7^m?2^n)(!2^n?0^l)] \\
&\longrightarrow \mathcal{E}[((\lambda.\lambda.())(!7^m?2^n))|7\Downarrow^w\ (!2^n?0^l)] \\
&\longrightarrow \mathcal{E}[((\lambda.\lambda.())(!7^m 2))|7\Downarrow^w\ (!2^n?0^l)] \\
&\longrightarrow \mathcal{E}[((\lambda.\lambda.())(()|2\Downarrow^m))|7\Downarrow^w\ (!2^n?0^l)] \\
&\longrightarrow \mathcal{E}[((\lambda.\lambda.())())|2\Downarrow^m\ |7\Downarrow^w\ (!2^n?0^l)] \\
&\longrightarrow \mathcal{E}[(\lambda.())|2\Downarrow^m\ |7\Downarrow^w\ (!2^n?0^l)] \\
&\longrightarrow \mathcal{E}[(\lambda.())|2\Downarrow^m\ (!2^n?7^l)|7\Downarrow^w] \\
&\longrightarrow \mathcal{E}[(\lambda.())(!2^n?7^l)|2\Downarrow^m\ |7\Downarrow^w] \\
&\longrightarrow \mathcal{E}[(\lambda.())(!2^n 7)|2\Downarrow^m\ |7\Downarrow^w] \\
&\longrightarrow \mathcal{E}[(\lambda.())(()|7\Downarrow^n)|2\Downarrow^m\ |7\Downarrow^w] \\
&\longrightarrow \mathcal{E}[((\lambda.())())|7\Downarrow^n\ |2\Downarrow^m\ |7\Downarrow^w] \\
&\longrightarrow \mathcal{E}[()|7\Downarrow^n\ |2\Downarrow^m\ |7\Downarrow^w]
\end{aligned}
$$

The effect of swap is not changing the context but generating three write effects.

Let us see the type of swap. Typed swap is

$$\mathrm{swp} = \Lambda\rho_1{:}\mathrm{region}.\Lambda\rho_2{:}\mathrm{region}.\lambda x{:}\tau_1^\rho.\lambda y{:}\tau_2^\rho.(\lambda.\lambda.\lambda.())(!0^l?x)(!x?y)(!y?0^l)$$

and type of it is

$$[l \leftarrow \rho] \vdash \mathrm{swp}{:}\ \Delta\rho_1{:}\mathrm{region}.\Delta\rho_2{:}\mathrm{region}.\tau_1^\rho \to \tau_2^\rho \to \mathbf{1}\ddagger W\rho \wedge R\rho_1 \wedge W\rho_1 \wedge R\rho_2 \wedge W\rho_2 \wedge R\rho$$

Since there is free variable l in swp (see the definition of fv) and l has type ρ , effects on region ρ can not be masked. It is justifiable because, there are possiblities that the location l apear outside of this term then the write effect can be observed there. On the other hand, another swap

$$\Lambda\rho_1{:}\mathrm{region}.\Lambda\rho_2{:}\mathrm{region}.\lambda x{:}\tau_1^\rho.\lambda y{:}\tau_2^\rho.\lambda w{:}\tau^\rho.((\lambda.\lambda.\lambda.())(!w?x)(!x?y)(!y?w))(\#[\tau][\rho]0)$$

has a type

$$\vdash \Delta\rho_1{:}\mathrm{region}.\Delta\rho_2{:}\mathrm{region}.\tau_1^\rho \to \tau_2^\rho \to \mathbf{1}\ddagger W\rho \wedge R\rho_1 \wedge W\rho_1 \wedge R\rho_2 \wedge W\rho_2 \wedge R\rho \wedge A\rho$$

and has no free variable. So, the effect mask inference rule can be applied and the result type is

$$\vdash \Delta\rho_1{:}\mathrm{region}.\Delta\rho_2{:}\mathrm{region}.\tau_1^\rho \to \tau_2^\rho \to \mathbf{1}\ddagger R\rho_1 \wedge W\rho_1 \wedge R\rho_2 \wedge W\rho_2$$

7 Record types in imperative system

Record type in functional system has a special modification operation as below.

$$\mathrm{incx}\,\{x = 3\} \mapsto \{x = 4\}$$

$$\mathrm{incx}\,\{x = 3,\ c = b\} \mapsto \{x = 4,\ c = b\}$$

This record modification is good to have especially in modeling object oriented calculation, but hard to type.

On the other hand, the same record modification can be implemented without typing difficulty in imperative system. We design the modifiable record with fields that value is a cell.

$$\{x = 3^n,\ c = b^l\}$$

Record modification can be implemented by field selection and ordinal assignment.

$$\mathrm{incx}(r) \stackrel{\mathrm{def}}{=} \mathrm{inc}(r.x)$$

The reduction example is

$$\mathrm{incx}\{x = 3^n\} \xrightarrow{*} ()|4\Downarrow^n$$

$$\mathrm{incx}\{x = 3^n,\ c = b^l\} \xrightarrow{*} ()|4\Downarrow^n$$

Since, the result of above two reduction are exactly identical, typing to the polymorphic incx is not difficult.

$$\mathrm{incx}\colon \{x\colon \mathrm{int}^\rho\} \to \mathbf{1}\ddagger W\rho$$

$$\{x\colon \mathrm{int}^\rho,\ c\colon \mathrm{bool}^{\rho'}\} \leq \{x\colon \mathrm{int}^\rho\}$$

8 Where comes from region?

Region notion of type system of i-calculus is just borrowed from type system of FX. Region notion is a kind of trick in the both imperative type systems. It is possible to allocate more than one cells in one region. However, there is no reason to do so willingly because it makes type inference less precise. For the precision of type inference, eacn cell should have own region.

$$V^n\colon \tau^n$$

However, the number of cells allocated during execution is not limited in general. So, static typing does not use a label itself as a region. Typing require limiting the number of region to finite. There are several solutions for this problem. An important thing is that the region notion is the mechanism to turn this problem out from the type system.

Here we show a list of candidate of those solutions.
Given:

$$(\lambda f.\cdots \#0 \cdots f() \cdots f() \cdots)(\lambda.\#0)$$

(1) Only one region in a program
$(\lambda f.\cdots \#[r]0 \cdots f() \cdots f() \cdots)(\lambda.\#[r]0)$

(2) regions for each occurrence of !. $(\lambda f.\cdots \#[r_1]0 \cdots f() \cdots f() \cdots)(\lambda.\#[r_2]0)$

(3) distinguish call sites
$(\lambda f.\cdots \#[r_1]0 \cdots f[c_1]() \cdots f[c_2]() \cdots)(\Lambda\rho.\lambda.\#[\langle \rho\, r_2\rangle]0)$

(4) etc.

9 Conclusions

We have developed i-calculus that is an imperative reduction system with FX like type system. The reduction uses write effect moving idea.

For future work, it will be important to analize properties of a program by using imperative typing information. For example, cell is inherently global in i-calculus and some locality can be analized by effect masking type inference. There are more localities that can not be caught by effect masking inference.

There are plenty of works on static analysis of imperative program. It would be also important to study the relation between imperative types and those static analysis.

References

[1] Lucassen, J. and Gifford, D: Polymortphic Effect Systems *Prec of the 15th Annual ACM Conference on Principles of Programming Languages*, pp. 47-57 (1988)

[2] Felleisen, M. and Friedman, D. P. : A Calculus for Assignments in Higher-Order Languages *Proc. Symp. on Principles on Programming Languages*, pp. 314-325 (1987)

[3] Felleisen, M.: λ-v-CS: An Extended λ-Calculus for Scheme *Proc. Conference on Lisp and Functional Programming*, pp. 72-85 (1988)

[4] Kazuo Otake: An imperative computatin model and its types for object orineted programming 「オブジェクト指向のための手続き的計算モデルと型」情報処理学会 記号処理 60-6・プログラミング 2-6 合同研究会 (1991)

[5] Kazuo Otake: A calculus with assignment for object oriented programming and identity of objects 「オブジェクト指向のための代入を持つ計算系と同一性」情報処理学会 第 44 回全国大会 講演論文集 (5) pp.33-34 (1992)

[6] Scott Danforth and Chris Tomlinson.: Type theories and object-oriented programming. *ACM Computing Surveys,* Vol.20, No. 1, pp.29-72, March 1988.

[7] Shivers, O. *The semantics of scheme control-flow analysis.* the first ACM SIGPLAN and IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation, June 1991.